Data oddania sprawozdania: 24.04.2018 r.
Autor: Jakub Majewski 238902
Prowadzący projektu: Dr inż. Dariusz Banasiak
Termin zajęć: wtorek TN 13:15-15:00

Sprawozdanie z projektu SDiZO

Zadanie projektowe nr 1

Badanie efektywności operacji na danych w podstawowych strukturach danych

I. Wstęp

Celem zadania projektowego było zrealizowanie własnych implementacji podstawowych struktur danych w języku C++ oraz zmierzenie złożoności obliczeniowej podstawowych operacji takich jak dodawanie, usuwanie i szukanie elementu w strukturze.

Zaimplementowane struktury i operacje na nich poddawane pomiarom czasowym:

A. `Table` - lista oparta na tablicy dynamicznej
   1. `push/push_back/push_front` - wstawianie elementu o konkretnej wartości na dowolnej pozycji/na końcu/na początku tablicy
   2. `pop/pop_back/pop_front` - usunięcie elementu z dowolnej pozycji/z tyłu/z przodu tablicy
   3. `find` - szukanie konkretnego elementu w tablicy

B. `List` - lista dwukierunkowa
   1. `push/push_back/push_front` - wstawianie elementu o konkretnej wartości na dowolnej pozycji/na końcu/na początku listy
   2. `pop/pop_back/pop_front` - usunięcie elementu z dowolnej pozycji/z tyłu/z przodu listy
   3. `find` - szukanie konkretnego elementu w liście

C. `MaxHeap` - kopiec binarny typu maksimum
   1. `insert` - wstawianie elementu o konkretnej wartości do kopca
   2. `extract_max` - usuwanie elementu o największej wartości z kopca
   3. `find` - szukanie konkretnego elementu w kopcu

D. `RedBlackTree` - drzewo czerwono-czarne
   1. `insert` - wstawianie elementu o konkretnej wartości do drzewa
   2. `remove` - usuwanie elementu o konkretnej wartości z drzewa
   3. `find` - szukanie konkretnego elementu w drzewie

Możliwe średnie złożoności obliczeniowe wyżej wymienionych struktur:

	Table	List	MaxHeap	RedBlackTree
*push_back/push_front	O(n)/*O(1) capacity>>size	O(1)	-	-
push	O(n)	O(1)+O(n)[search]	-	-
insert	-	-	O(1)	O(log n)
pop_back/pop_front	O(n)	O(1)	-	-
pop	O(n)	O(1)+O(n)[search]	-	-
extract_max	-	-	O(log n)	-
remove	-	-	-	O(log n)
find	O(n)	O(n)	O(n)	O(log n)

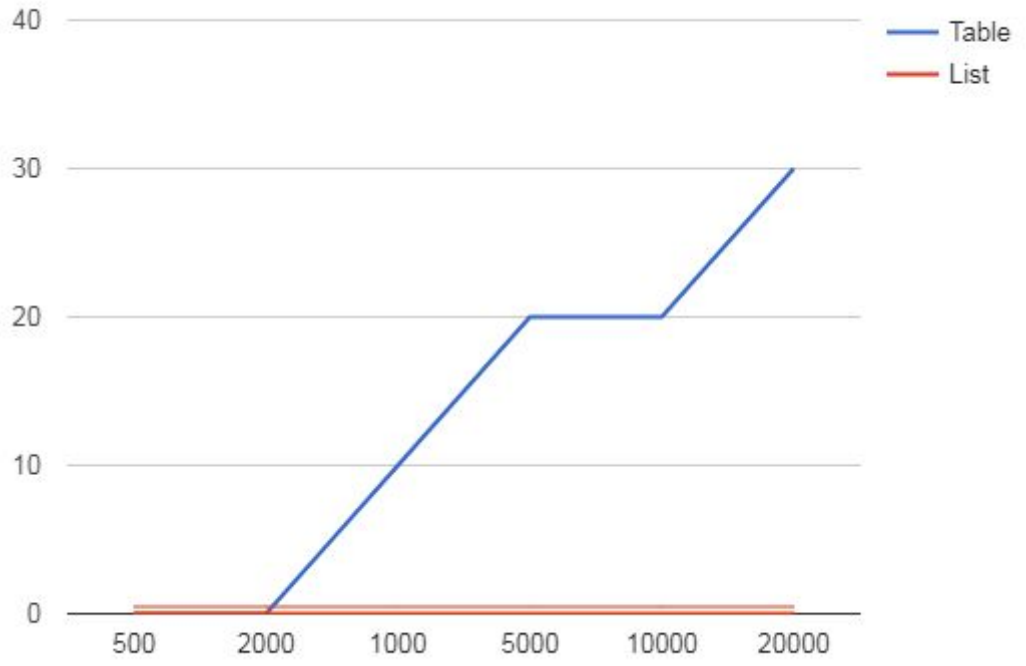
II. Założenia:

1. **Implementacja struktur:** Wszystkie struktury włącznie z programem testującym i funkcjami pomiaru są zaimplementowane przy użyciu języka C++ (standard C++11 lub wyższy). Jakiekolwiek biblioteki zewnętrzne nie są załączane, a zawartość standardowej biblioteki STL nie jest wykorzystywana w bezpośrednich implementacjach struktur.
2. **Wielkość struktur:** Ilość elementów zawsze jest równa maksymalnej pojemności struktury. W przypadku tablic maksymalny rozmiar [capacity] zawsze jest równy ilości przechowywanych elementów.
3. **Przechowywany typ danych:** Wszystkie elementy przechowywane w strukturach są typu 4-ro bajtowego całkowito-liczbowego (w C++ jest to typ `int`).
4. **Sposób generowania elementów:** Wszystkie struktury zawierają elementy wygenerowane losowo przy pomocy generatora Mersenne Twister Engine, którego implementacja jest dostępna w standardowej bibliotece C++ w bibliotece `<random>`.
5. **Sposób pomiaru czasu:** Czas mierzony jest z użyciem funkcji dostępnych w standardowej bibliotece C++ w bibliotece <chrono>.
6. **Wynik końcowy:** Czas końcowy jest reprezentowany w postaci średniej ze 100 prób. Na początku każdej próby struktura zawiera dokładnie taką samą ilość elementów jak podczas próby wcześniejszej (przed wywołaniem operacji mierzonej), ale same wartości tych elementów są za każdym razem generowane losowo. Czas jest mierzony z dokładnością do 0,001 mikrosekundy (odmierzany w nanosekundach i dzielony przez 1000).

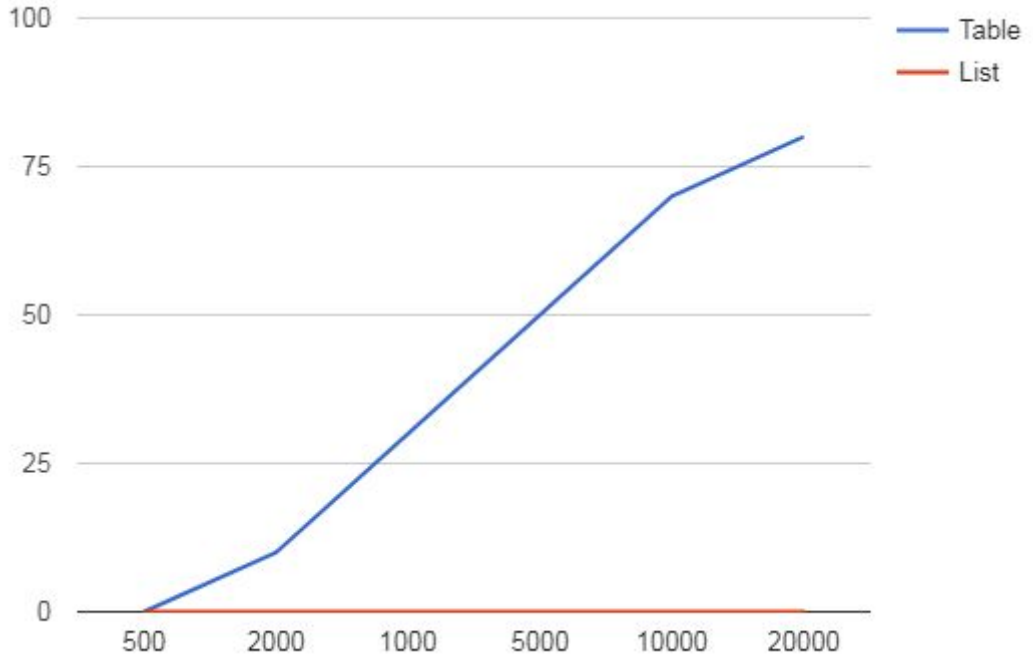
III. Wyniki testów

Każda tabela wyników zawiera czas wykonywania jednej operacji zależny od struktury oraz początkowego rozmiaru struktury [`capacity`] przed wykonaniem tej operacji.
Wszystkie wyniki pomiarów są średnią ze 100 prób reprezentowane dokładnością do 0,001 mikrosekundy. Wszystkie dane z tabel pochodzą z wygenerowanych uprzednio plików `vector_results.txt` oraz `tree_results.txt`.

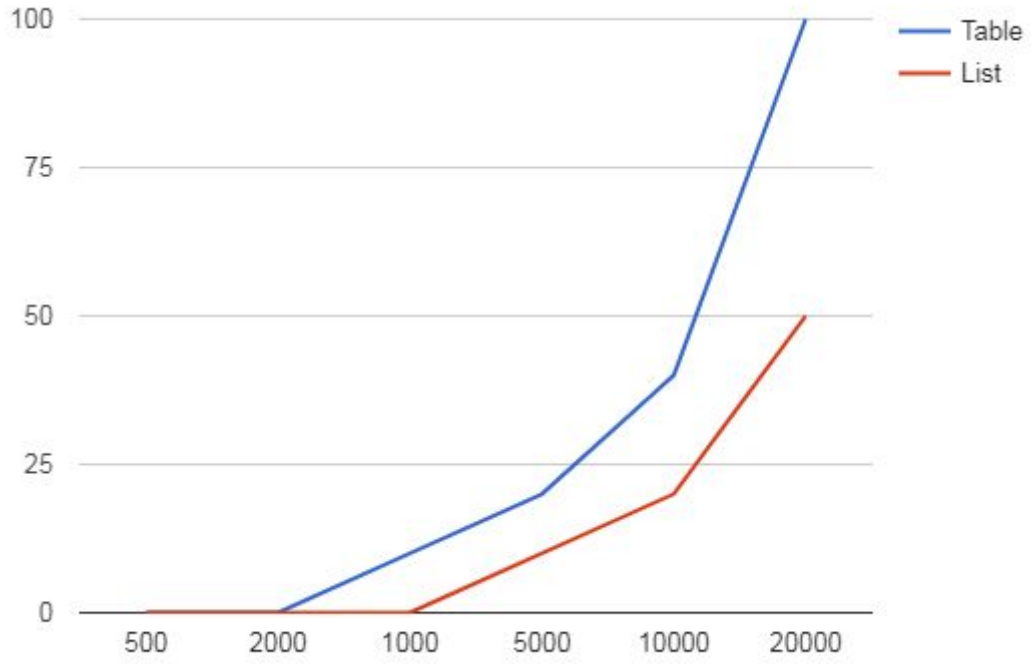
push_back		
capacity	Table	List
500	0	0
1000	10.001	0
2000	0	0
5000	20	0
10000	20	0
20000	30.002	0



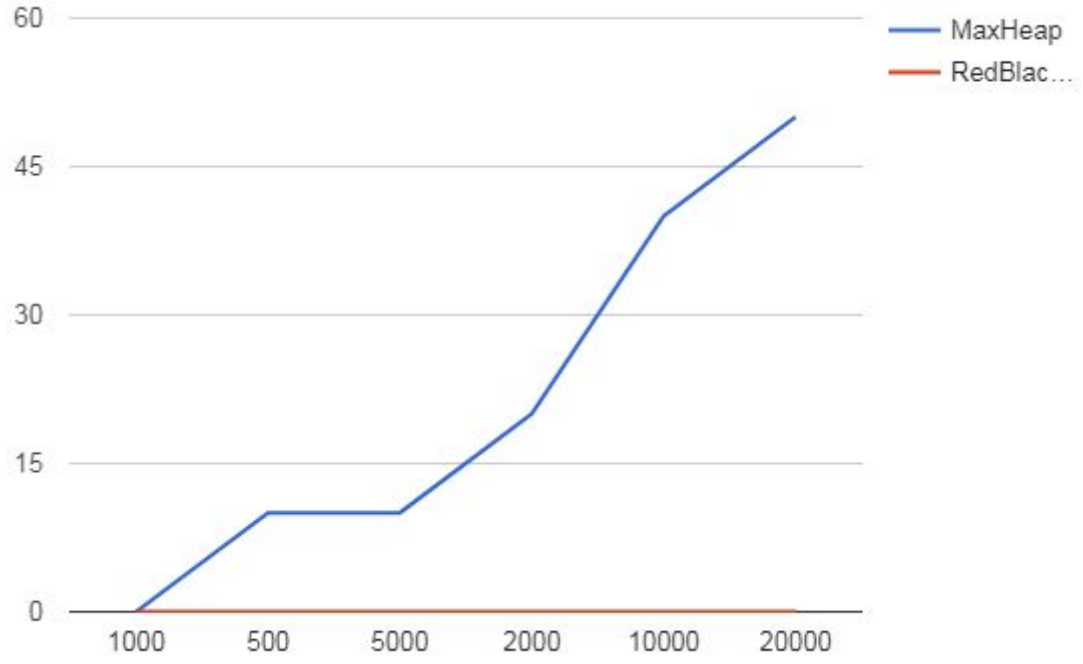
push_front		
capacity	Table	List
500	0	0
1000	10.001	0
2000	30.001	0
5000	50.004	0
10000	70.003	0
20000	80.004	0



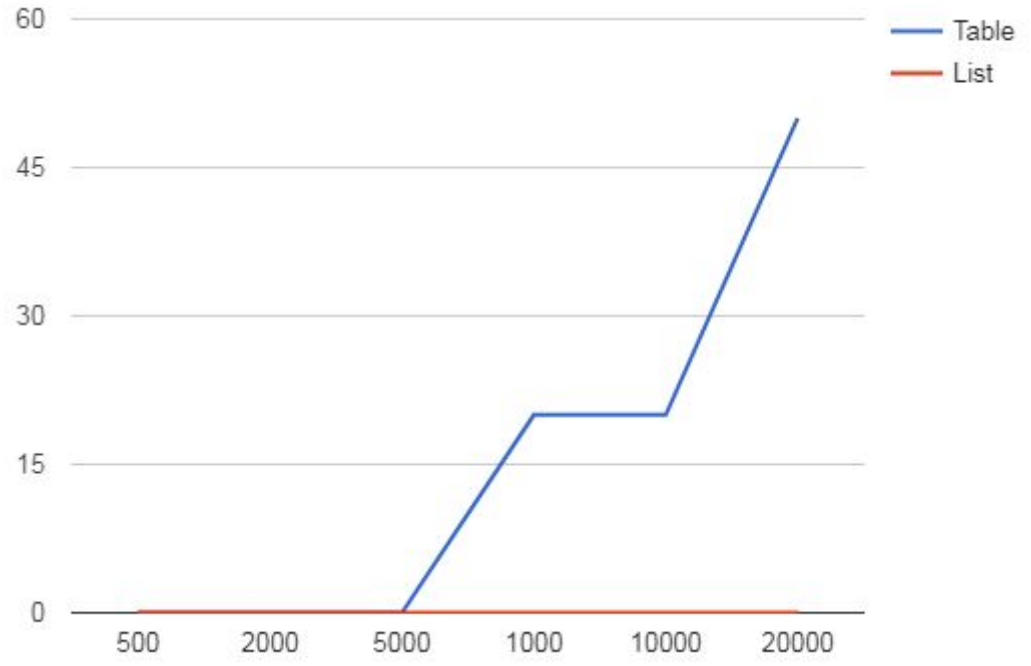
push		
capacity	Table	List
500	0	0
1000	0	0
2000	10.001	0
5000	20.001	10
10000	40.003	20.001
20000	100.004	50.004



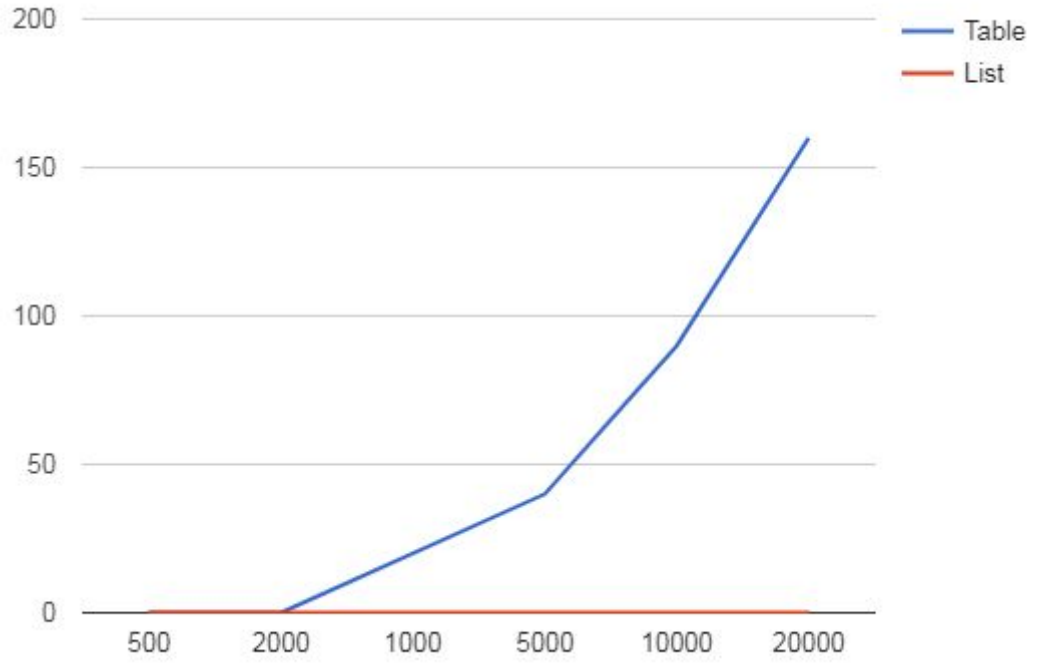
insert		
capacity	MaxHeap	RedBlackTree
500	10	0
1000	20.001	0
2000	0	0
5000	10	0
10000	40.003	0
20000	50.003	0



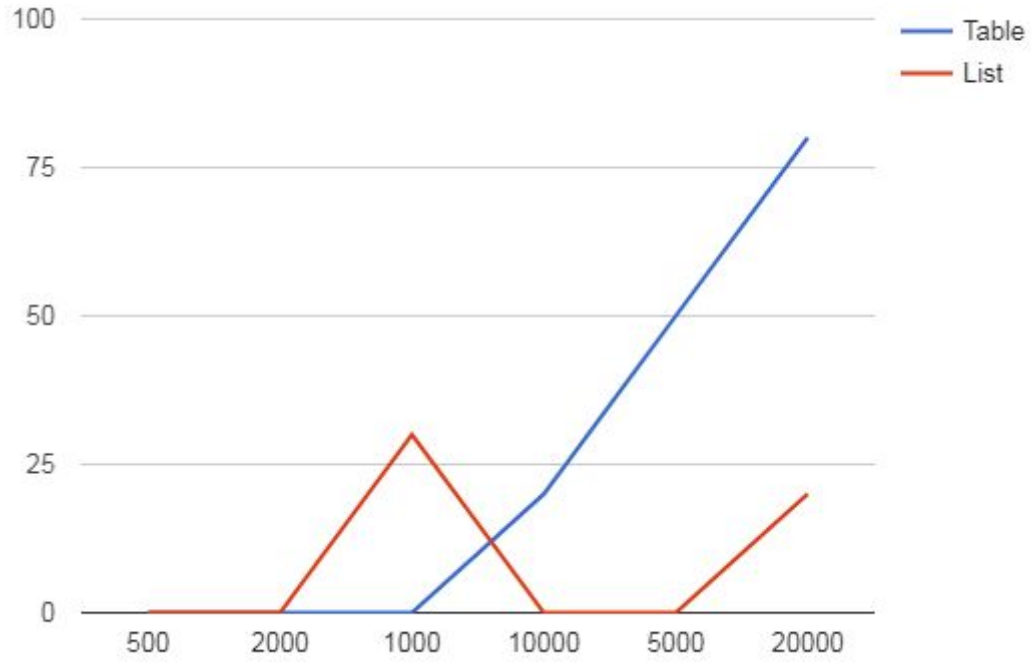
pop_back		
capacity	Table	List
500	0	0
1000	0	0
2000	20.001	0
5000	0	0
10000	20.002	0
20000	50.002	0



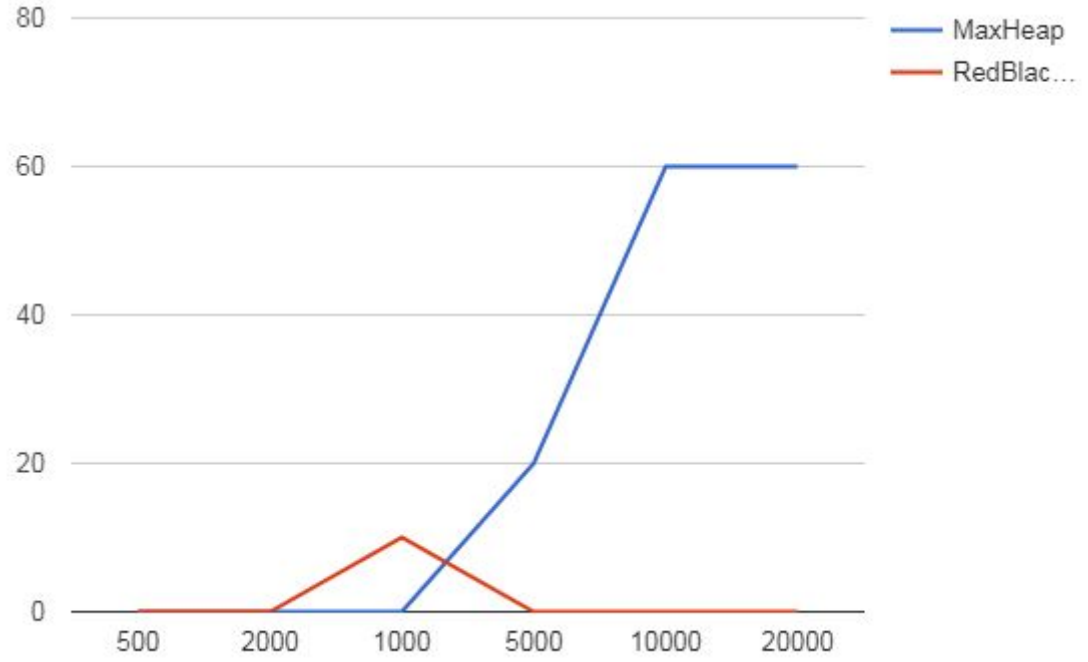
pop_front		
capacity	Table	List
500	0	0
1000	0	0
2000	20	0
5000	40.002	0
10000	90.006	0
20000	160.011	0



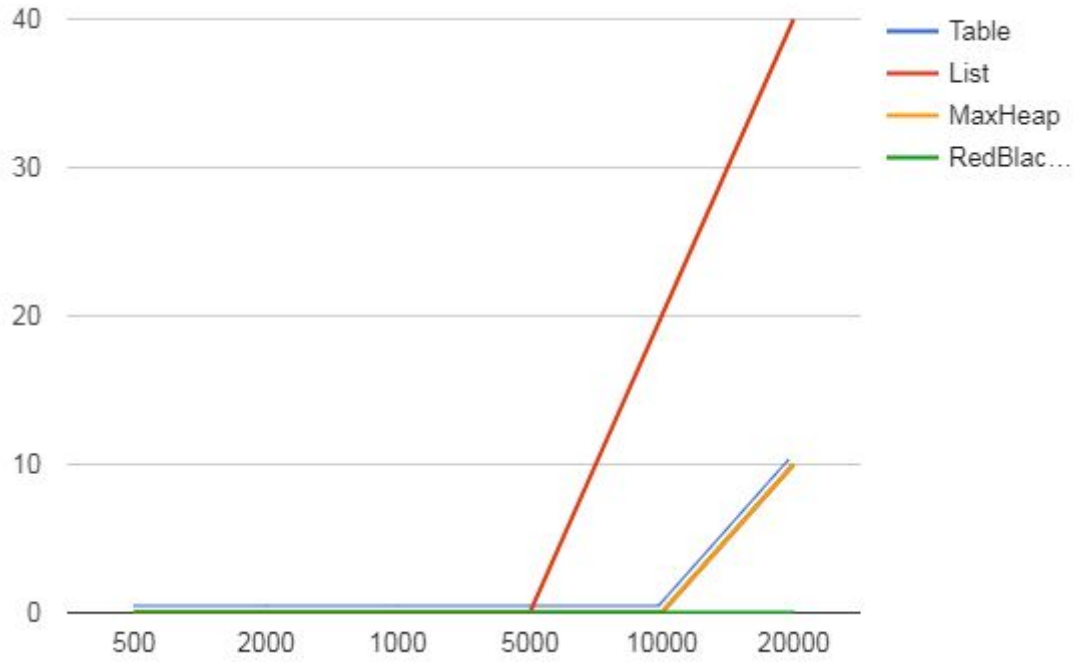
pop		
capacity	Table	List
500	0	0
1000	0	0
2000	0	30
5000	50.001	0
10000	20.002	0
20000	80.004	20.002



extract_max (MaxHeap) / remove (RedBlackTree)		
capacity	MaxHeap	RedBlackTree
500	0	0
1000	0	0
2000	0	10
5000	20.001	0
10000	60.002	0
20000	60.002	0



find				
capacity	Table	List	MaxHeap	RedBlackTree
500	0	0	0	0
1000	0	0	0	0
2000	0	0	0	0
5000	0	0	0	0
10000	0	20.002	0	0
20000	10	40.001	10	0



IV. Wnioski

Analizując otrzymane wyniki można wyciągnąć następujące wnioski:

1. Jeżeli potrzebujemy struktury zachowującej kolejność elementów, która często będzie modyfikowana w postaci dodawania elementów na początku bądź na końcu (idealnym przykładem jest baza danych działająca online) to zdecydowanie najlepszym wyborem będzie lista dwukierunkowa.
2. Jeżeli zależy nam na przechowywaniu danych i pozyskiwaniu z nich największego (bądź najmniejszego) elementu wówczas najbardziej sensownym wyborem będzie kopiec binarny. Trzeba mieć jednak na uwadzę, że usuwanie największego (bądź najmniejszego) elementu jest całkiem czasochłonną operacją.
3. Jeżeli chcemy przechować jakieś elementy po danym kluczu i po podaniu klucza mieć do nich szybki dostęp wówczas drzewo czerwono czarne wydaje się najlepszym wyborem.
4. Jeżeli jesteśmy w stanie określić mniej więcej ilość elementów potrzebnych do przechowania i chcemy mieć do nich szybki dostęp poprzez indeks wtedy idealną strukturą będzie lista oparta na tablicy dynamicznej.

V. Kod źródłowy

Kod źródłowy programu składa się z trzech głównych części zawierających:

1. Implementacje struktur:
    - `table.hpp` - lista oparta na tablicy dynamicznej
    - `list.hpp` - lista dwukierunkowa
    - `heap.hpp` - kopiec binarny typu maksimum
    - `red_black_tree.hpp` - drzewo czerwono czarne
2. Specjalne klasy “testerów” wykonujące testy i pomiary:
    - `vector_tester.hpp` - pozwala testować listy
        - `push/push_back/push_front`
        - `pop/pop_back/pop_front`
        - `find`
    - `heap_tester.hpp` - pozwala testować kopiec binarny
        - `insert`
        - `extract_max`
        - `find`
    - `tree_tester.hpp` - pozwala testować struktury drzewiaste
        - `insert`
        - `remove`
        - `find`
3. Klasa `Interpreter` symulująca interpreter własnych komend umożliwiający testowanie metod zaimplementowanych struktur w czasie działania programu (run-time). Umożliwia również ładowanie danych do struktur z plików.
    - `interpreter.hpp`

Ponad to w pliku `main.cpp` znajduje się globalna funkcja `measure_time` ułatwiająca korzystanie z “testerów” oraz główna funkcja `main`.

Przyjęty style-guide:

- notacja `camelCase` - nazwy zmiennych i składowych klas (wyjątki w przypadku jednoliterowych zmiennych)
- notacja `PascalCase` - nazwy klas i aliasów
- notacja `lower_snake` - nazwy metod, funkcji, przestrzeni nazw i plików
- dla ułatwienia wszystkie składowe i metody klas (z małymi wyjątkami) są publicznie
- kod poza plikiem `main.cpp` składa się wyłącznie z plików nagłówkowych, jest to zabieg świadomy w celu zachowania ładu przez wzgląd na użycie klas i metod szablonowych.

Wykorzystane biblioteki z biblioteki standardowej C++:

- `iostream`
- `fstream`
- `sstream`
- `iomanip`
- `vector`
- `chrono`
- `random`

Kod źródłowy jest dostępnych pod linkiem: `github.com/MetRiko/SDiZO-project`

Pomoce naukowe:

1. `opendatastructures.org/ods-cpp.pdf`
2. `math.uni.lodz.pl/~horzel/ZA_2008/3_drzewa_RB.pdf`
3. `en.wikipedia.org`